Static Analysis of information flow for Python
Case study: Verification of the back-end of an e-voting system

João de Araujo Correia Neto Lopes

Thesis to obtain the Master of Science Degree in
Telecommunications and Informatics Engineering

Supervisors:  Prof. Jan Gunnar Cederquist
              Prof. Ana Gualdina Almeida Matos

Examination Committee
Chairperson: Prof. Ricardo Jorge Fernandes Chaves
Supervisor: Prof. Jan Gunnar Cederquist
Member of the Committee: Prof. António Paulo Teles de Menezes Correia Leitão

November 2020
Dedicated to someone special...
Acknowledgments

I would like to thank my family and friends for being there to provide all the help when I most needed it. I would also like to thank my supervisors Ana Almeida Matos and Jan Cederquist who were always available to provide help and guidance when necessary as well as Instituto Superior Técnico for providing me with the necessary conditions to finish this work.
Resumo

Com a evolução da tecnologia, o desenvolvimento de software aumentou significativamente e, como consequência, a necessidade de programadores de software nunca foi tão elevada. À medida que essa necessidade aumenta também aumenta a probabilidade da segurança do software produzido passar para segundo plano.

Num mundo onde uma maior dependência no software implica mais métodos que os atacantes podem usar, a afirmação anterior é uma declaração muito perigosa. Uma das linguagens de programação mais usada é Python. Python é amplamente usado para implementar o lado do servidor de aplicações, pois várias estruturas estabelecidas foram criadas para esta linguagem, bem como um grande número de bibliotecas de terceiros para o programador usar. Python é considerada uma linguagem dinâmica, o que significa que algumas operações que poderiam ser realizadas antes da execução agora o são em tempo de execução.

Para realizar uma análise automática do código que fornece fortes garantias de segurança, propomos uma ferramenta baseada no fluxo de informações capaz de analisar e detectar fluxos ilegais em termos de confidencialidade e integridade, exigindo apenas que o programador anote o seu código e que forneça os níveis de segurança que julgar necessários.

O principal caso de teste deste trabalho será a análise de um sistema de votação eletrônica. Como o voto eletrônico depende da tecnologia, isso implica que este sistema herda também todas as suas falhas e problemas de segurança. A votação eletrônica é o caso de teste perfeito, pois é altamente dependente das propriedades que analisamos, tanto a confidencialidade quanto a integridade.

Palavras-chave: Análise Python, Análise Estática, Information Flow
Abstract

With the evolution of technology, software development has highly increased and as so, the need for software developers has never been higher. As that need increases, so does the probability that security of the produced software takes a toll in terms of priority. And in a world where more dependance on software also translates into more methods attackers can use, that is a very dangerous statement. One of the most used programming languages is Python. Python is widely used to implement the server-side applications as several established frameworks have been created for it as well as a high number of third party libraries for the programmer to use. Python is considered a dynamic language, which means that some operations that could be performed before execution are now done so at run-time.

In order to perform an automatic analysis on the code that provides strong security guarantees, we introduce a tool based on information flow that is capable of detecting illegal leaks in terms of both confidentiality and integrity, while only requiring the programmer to type annotate its code as well as providing the security levels that he deems necessary.

The main test case of this work will be the analysis of an Electronic voting system, also known as e-voting. Since e-voting relies on technology, it consequently results in it inheriting all its flaws and problems security-wise. E-voting is the perfect test case since it highly emphasizes on the properties that we analyze, both confidentiality and integrity.

Keywords: Information flow, static analysis, Python analysis
# Contents

Acknowledgments .................................................. v
Resumo .................................................. vii
Abstract .................................................. ix
List of Tables .................................................. xiii
List of Figures .................................................. xv
Nomenclature .................................................. 1
Glossary .................................................. 1

1 Introduction .................................................. 1
  1.1 Motivation .................................................. 1
  1.2 Objectives .................................................. 2
  1.3 Contributions .................................................. 2
  1.4 Thesis Outline .................................................. 3

2 Background .................................................. 5
  2.1 Protocol Specification .................................................. 5
  2.2 Static vs dynamic analysis .................................................. 7
  2.3 Information Flow .................................................. 7

3 Related Work .................................................. 9
  3.1 Verification of e-voting protocols .................................................. 9
    3.1.1 Formal-languages-Applied pi-calculus .................................................. 9
    3.1.2 Type-based .................................................. 10
  3.2 Implementation verification of e-voting systems .................................................. 10
  3.3 Security Tools for Python .................................................. 11
    3.3.1 Taint Mode .................................................. 11
    3.3.2 Symbolic Execution .................................................. 12
    3.3.3 Type checking .................................................. 12
    3.3.4 Information Flow .................................................. 14
  3.4 Information Flow for full fledged languages .................................................. 14
List of Tables
List of Figures

4.1 Inheritance. ....................................................... 18
4.2 Security Lattice. ............................................... 19
4.3 Catbox dilemma. ............................................... 21
4.4 AST representation .............................................. 22
5.1 The stages of the tool. ......................................... 23
5.2 Dictionary cell. .................................................. 37
5.3 Dictionary .......................................................... 39
Chapter 1

Introduction

1.1 Motivation

With the evolution of technology, software development has highly increased and as so, the need for software developers has never been higher. In the current time, the whole world is depending on software, from small businesses to multinational companies. Among other factors, the fact that the needs and requirements of software are ever-evolving translates into ensuring that there won’t be a shortage of demand for software developers in the foreseeable future. As that need increases, so does the probability that security of the produced software takes a toll in terms of priority. And in a world where more dependance on software also translates into more methods attackers can use, that is a very dangerous statement. It is natural that as software development flourishes, so does the market that wants to exploit it, so the need for security should be getting higher in our priority list, not lower. Even if the developer takes security into consideration, he will most likely use third party libraries for which he may not even have access to the source code.

Python is widely used to implement the server side of applications as several established frameworks have been created for it as well as a high number of third party libraries for the programmer to use. Considering how often Python is used it is definitely important to develop tools that can analyze it. Python is considered a dynamic language, which means that some operations that could be performed before execution are now done so at run-time. One example would be the possibility to change a variable's type as the program in being executed. As a consequence, analyzing such a language in regard to its security can be challenging.

To address the former challenges we propose the usage of information flow analysis. In information flow[1] analysis, the variables in a program are usually divided into two security levels, high and low. If we are guaranteeing confidentiality these can correspond to, respectively, secret information, and public information. If on the other hand we are ensuring integrity, these can be represented as tainted and untainted information. If the changes in a variable from one level spread to the other level we may have an illegal flow depending on whether we are ensuring confidentiality or integrity. If we are evaluating a program’s confidentiality changes in high level variables cannot be visible in low level variables, on the
other hand if we are evaluating the integrity of a program we can’t allow flows from low level variables to high level variables.

This work will focus its attention on the TAGUS e-voting system, which was developed in IST(Instituto Superior Tecnico) as a voting solution that is suitable for Universities and other similar organisations. It is based on the H-Belenios protocol[2], that is an adapted version of Belenios[3] that also accounts for the traditional method of voting that involves voting through paper ballots. In order for the e-voting system to be useful it is essential that the security properties and assurances provided by the protocol are verified. Here, we briefly mention some of the most important ones. It is essential that the votes remain anonymous, which means that the voter can’t be associated with the corresponding vote. Auditability is also fundamental, the possibility for every voter to check if his vote was processed correctly. Another important property would be Confidentiality, which means that the vote content cannot be inferred until the final count. And last but not least Integrity, only authenticated voters can vote, and they can only do so once. As both confidentiality and integrity are vital for this e-voting system it supplies the perfect test case for our tool.

The aim of this work is to make a static Python analysis tool which is based on information flow. The solution involves the elaboration of a tool that allows for illegal information flows to be detected while only requiring the programmer to type annotate its code as well as providing the security levels that he deems necessary.

### 1.2 Objectives

This work addresses the fact that software is exposed to a number of threats from various sources. Its importance in society given the fact that software development is only increasing, ensures the need to make sure that software is secure.

This goal involves more specifically:

- **Creation of a static analysis tool that enforces information flow for Python.**

- **Ensuring that the server-side of the TAGUS e-voting system is secure.**

### 1.3 Contributions

Created a static analysis tool based on information flow that uses type annotations and a type checker to address most types of illegal flows in regard to two properties, integrity and confidentiality. This tool was originally created specifically for the code related to the server side of an e-voting system but was later extended to be able to apply information flow to any Python program.
1.4 Thesis Outline

The paper is structured in the following manner. We first start by introducing some background that the reader is required to know in the Section 2, which includes the knowledge needed of the e-voting protocol that will be analyzed, the relevant implementation details like the language and the essential aspects of the specification that the implementation must guarantee. We then proceed to the related work in Section 3 which includes the former means of verification of e-voting protocols and systems, the most common external and internal vulnerabilities that the server is subject to and the various techniques to fight them, and ends with the existing solutions to validate sanitization. Section 4 would address the model which will describe the solution and the idea behind it. Its implementation will be explained in Section 5. Leaving the Section 6 for validation and ending with the Conclusion in Section 7.
Chapter 2

Background

In this chapter we introduce some important notions that are vital to understand how to guarantee that the server is secure. In the first section we shed some light on the protocol itself. Section 2.2 will discuss and compare dynamic analysis with static analysis in the context of our system and Section 2.3 will supply the base knowledge needed in regards to information flow.

2.1 Protocol Specification

The protocol specified here is the one already mentioned earlier, H-Belenios. It is a hybrid of traditional paper voting and Belenios. Belenios itself is an improved version of Helios[4]. The focus of this section will be on analyzing the protocol in a more detailed manner emphasizing on the most important points regarding the server-side of the protocol. Even though the protocol discussed is H-Belenios, the priority when describing its main details will be on the Belenios protocol since the traditional paper one does not present the vulnerabilities that will be discussed further ahead.

We start by introducing some key acronyms that will be used throughout the explanation of the protocol.

- SA- Server administrator - the person who is responsible for creating and managing the election.
- VS- Voting server - server in which the main operations regarding the election happen.
- CS- Credential server - server whose main objective is to generate and distribute the credentials for the voters of the election.
- SBB - Smart Bulletin Board - public place that the voters/trustees use to check if everything was done correctly.
- TL - trustee list - List of people whose purpose is to encrypt and decrypt the ballots.
- EVL - Eligible voter list
• Zero knowledge proofs - proofs that attest to the correctness of the encryption without having to witness the process.

H-Belenios is divided in 4 phases:

Set up phase. The first step of the election is for the SA to send the election parameters to the VS. These parameters include the name of the election, the description of the election and the start and end dates of it. Afterwards the VS publishes the information in the SBB. Besides this information, it also sends it the TL. The trustees will then generate their own pair of ElGamal keys. The public key is sent to the VS by every trustee, it is then aggregated and put on the SBB so that they can be sure the public keys used were valid and the private is kept in secret by the trustees. If not for this measure the VS could potentially add new votes at will, but since every private key will be necessary to decrypt the ballots and only the trustees have access to them that is close to impossible as it would mean all of the trustees would be corrupt. The VS makes sure the keys that are generated are valid by analyzing the zero knowledge proofs that are sent with those. With this method the server does not have knowledge of the keys but is qualified to validate them. Finally, the SA will also send the EVL to the VS.

Voting phase. The next phase is the voting one which takes place between the two dates set by the SA in the first phase. Before being able to vote, the voters in the EVL must register in the CS. To do so each voter must first contact VS who then sends to the CS the voter's identifier and the corresponding parameters generated during the first phase. Subsequently the CS generates the key pair and sends the private key to the voter by email and the public key to the VS. The private key will be used to sign the ballot by the voter and the public to check the signature by the VS. This measure ensures that the voter is authentic. The ballot consists of the encrypted answer, encrypted using the public election key, a digital signature using the private key mentioned earlier, and the zero knowledge proofs. After the ballot is created, a hash is computed and stored by the voter. VS then receives the ballot, checks the zero knowledge proofs, saves the ballot, computes the hash over the ballot and divulges the hash through the SBB. The voter will then check if the hashes match to make sure his ballot is accounted for.

Tally phase. After the voting phase is terminated all the valid ballots are then aggregated, thanks to their homomorphic properties, and sent to each trustee. Homomorphic encryption is a type of encryption that allows computation on encrypted data, that generates an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the decrypted data. The goal of homomorphic encryption is to allow computation on encrypted data. The fact that it isn’t possible to decrypt the tally without every trustee ensures vote privacy since the ballot of an individual voter is never decrypted. After each trustee decrypts its part, it needs to add a zero knowledge proof so that the VS can be sure the decryption is valid. After every decryption is made, and the zero knowledge proofs analyzed, the VS
publishes them on the SBB.

Audit phase.

This phase was already mentioned in the prior phases, its goal is for the voters to check if their ballot and for the trustees to check if the encryption was valid. The first goal is verified by comparing the hashes that are published in the VS with their own. For the following one to be verified, all the trustees must compare their part of the key with the one published by the VS in the set up phase.

Needless to say that the auditioning depends on the quantity of both voters and trustees that follow the process mentioned earlier, the higher the percentage, the more secure will the election be.

2.2 Static vs dynamic analysis

The discussion between static and dynamic analysis and when to use one or the other is always vital when considering the security of a program. While dynamic analysis is normally used in an environment where a little bit of overhead is tolerated and the program can be stopped if there is a vulnerability detected, static analysis thrives on the fact that it can detect the errors prior to the execution even though it may introduce false positive results depending on the tool used.

In the context of the protocol being analyzed, both static and dynamic analysis can be applied.

Since the election cannot be stopped midway there is the need to apply static analysis for the core process of the server. That process cannot stop even if a vulnerability is detected, therefore the vulnerability in that case, must be detected earlier, prior to execution.

This being said it's essential to check if the implementation of the protocol allows for dynamic analysis to be used on the isolated "threads/processes" that are at risk and can be stopped at runtime without endangering the functioning of the whole server.

Given that there is total access to source code of the server there is no restriction in terms of which static tools may be applied.

When considering the nature of the protocol, its priority is focused not in efficiency but in making sure there are no confidentiality leaks and the integrity of the data remains accurate. Therefore it is of no consequence if limited overhead is introduced in the program due to dynamic analysis techniques. For example, it does not matter if the vote is only put in the bulletin board a few minutes after it is sent but it is vital that the trustee cannot divulge its private key or the integrity of the vote is lost.

2.3 Information Flow

In information flow[1] analysis, the variables in a program are usually divided into two security levels, high and low. If we are guaranteeing confidentiality these can correspond to, respectively, secret information, and public information. If on the other hand we are ensuring integrity, these can be represented as tainted and untainted information. If the changes in a variable from one level spread to the other level we may have an illegal flow depending on whether we are ensuring confidentiality or integrity. If we
are evaluating a program’s confidentiality changes in high level variables cannot be visible in low level variables, on the other hand if we are evaluating the integrity of a program we can’t allow flows from low level variables to high level variables.

Information flow is usually studied under one of two levels of precision. It can deal exclusively with explicit flows, which as the name implies, happens when a secret is explicitly exposed to a variable that can be seen by the public. And it can also address implicit flows, which happens when the execution arrives at a conditional statement and is divided into two branches, performing publicly observed side effects depending on which branch is taken [5].

Non-Interference is a property that ensures that every low input results in the same low output, no matter the difference in high level variables. Which means there cannot be anything inferred about the high level input through the observable output (low level variables).

Although theoretically this property would have to always be present, in real world applications it can be too restrictive to achieve any kind of useful result. It is therefore important that the mechanism we choose is able to allows us to, in a controlled manner, release as much information as possible without making it possible for the attacker to infer anything about the system. The application of this notion allows for more flexibility despite having to ensure the properties stated before.

When analyzing a program from a security standpoint it is very important that we have the possibility of intentionally lowering the security level of specific information so that the information can flow but in a controlled way so that a possible attacker cannot infer anything useful from the change made. This concept is called declassification. It is vital that this is done in a robust way, so when applying this concept, the programmer must think about several aspects. What information is being declassified? Who will be able to access the information after the fact? Where did it occur and when? [6]

It is also important to be able to increase of the integrity of information. There will be a time when we want to increase the integrity of a piece of information, for example when a variable just went through a sanitization function and is therefore safe to use it. If not for endorsement, a piece of information wouldn’t be able to change in the context of its security level and the example supplied earlier wouldn’t be able to be dealt with.

In light of the former two paragraphs, the ideal is not proving that a program complies with the Non-Interference property, because in practice that would be too restrictive, but instead proving it complies with a more flexible and more useful version of that property.
This section addresses the previous work done on the verification of e-voting protocols and systems, followed by the various types of security tools designed to fight the most common vulnerabilities for Python and ending with Information Flow approaches regarding other programming languages.

3.1 Verification of e-voting protocols

E-voting protocols [2] are very hard to analyze and verify mainly due to its high dependency on the use of cryptographic tools, which makes e-voting protocols very difficult to model. A work-around to this problem is to use an abstract model of the e-voting protocol and analyze it against formally stated properties.

Recently highlighted inadequacies of implemented systems have demonstrated the importance of formally verifying the underlying voting protocols.

3.1.1 Formal-languages-Applied pi-calculus

Formal-languages have since long been used by mathematicians, they are defined as a set of symbols with a specific set of rules attached to them. Its alphabet consists of these symbols, and the rules define how the strings of symbols can be grouped and built in order to be well-formed. These are also called words. As stated before formal-languages are very valuable when dealing with the verification of e-voting protocols. This gives you the possibility of representing processes, parallel composition of processes, synchronous communication between processes through channels, creation of channels, replication of processes, and non-determinism. The applied pi-calculus is nothing but a simple extension of it, in which values can be formed from names via the application of built-in functions, subject to equations, and be sent as messages [7].

In [8] the focus is on the three following properties, vote-privacy, receipt-freeness and coercion-resistance. Vote-privacy, like the name implies, means that the choice of the voter cannot be revealed to anyone. Receipt-freeness ensures that a voter cannot prove to anyone the contents of his vote. Coercion-resistance is the third and strongest of the three privacy properties. It says the voter wont be
able to be associated with his vote, even if the voter cooperates with the attacker during the election process.

The applied pi-calculus is once more used since it is useful when modeling such protocols. This formal language has the advantages of being based on well-understood concepts. The idea is to model an infrastructure to assess e-voting protocols in regard to the properties defined earlier.

The formalized properties defined in the previous paper are here [9] used in two e-voting protocols.

3.1.2 Type-based

In contrast to the previously mentioned solutions, [10] proposes a type checker that acts based on pre and post conditions with the goal of verifying two vital properties in the context of e-voting, vote privacy and verifiability.

Instead of being based on theorem proving, this approach chooses to focus on type systems that at first seems less precise compared to theorem provers. Although it may look that way, since they are structured over code instead of abstract-terms, they enable a more in-depth analysis of the cryptographic schemes used by e-voting like homomorphic encryption that would otherwise be impossible to address.

3.2 Implementation verification of e-voting systems

In the former section we talked about the various approaches to deal with the verification of the specification of e-voting protocols, here we are going beyond the specification of the protocol and analyze how the implementation itself is verified.

Even though the system analyzed in [11] is relatively simple for an e-voting protocol it still has the same challenges that come with the use of cryptographic functions. It is this fact that makes the verification of e-voting very hard, if not impossible, to perform through automated tools. The solution proposed to address this problem is a hybrid tool that still allows for some degree of automatization through verification tools based on checking non-interference properties. The difference is that for the cases that require a more detailed attention the interaction with human beings must be stronger, e.g. through theorem provers. The idea is to combine automated analysis which is normally faster but more imprecise with a more interactive approach that relies more on human interaction and is therefore slower, which means the goal is to use the interactive analysis as few times as possible, only when the automated one fails.

Civitas [12] is an e-voting system that guarantees confidentiality and integrity as a result of being built over JiF, JiF, based on JiF which is an extension of Java that provides security features, is a security-typed language that uses information-flow policies as annotations in programs to enforce these two properties. Even though Civitas has multiple entities which have to cooperate together but do not trust each other, like all e-voting systems, it was shown that it is still able to uphold both integrity and confidentiality properties[14].
3.3 Security Tools for Python

Having covered the existing work in regard to e-voting, both protocol-wise and implementation-wise, we move on to the various methods and tools available for Python. This section will be focused on Python since the main objective of this work is to ensure the security of the server part of the e-voting system. This section will start by targeting taint mode tools while providing a brief explanation on what the concept means. This method is one of the most important methods for us since our work is based on it. Subsequently we will cover Symbolic execution with a brief explanation of its advantages and disadvantages. We will then dive into type checking, which is a huge component of our solution, and finally end the section with information flow.

3.3.1 Taint Mode

The idea behind taint mode is that any variable that can be accessed by an external user can be a security liability. Therefore a list of those variables is compiled, including the variables influenced by those mentioned earlier. If any of the variables in the list are used as input to execute a potentially dangerous function, for example a SQL injection, the programmer is advised to sanitize the input before using it.

This taint analysis can be done statically or dynamically. Static analysis is done based on the source code without executing it, which means it does not generate additional overhead. Although this is useful, it also brings several limitations. Since it is done before the program is executed it does not have access to the input given. Another important drawback of static analysis is the fact that it generates a high number of false positives, which means it flags input that would be harmless just because of the source being unknown or untrusted.

But the most important disadvantage is the fact that static analysis is useless against executable code in the format of a string which is possible in python[15] if certain built-in functions are used. Dynamic analysis has none of the disadvantages mentioned earlier that results in it being chosen over static analysis in regard to taint analysis.

In a language like python third-party modules are essential and often used. The impossibility of accessing the source code of the modules used makes it very hard to analyze. Invisitype [16] makes it possible by encapsulating object-oriented safety-checking rules as policy classes. They do so by taking advantage of the multiple inheritance property. Another advantage of this approach is that only the instances at risk are subject to overhead since these are determined at runtime through the virtual method dispatch. But like most dynamic approaches, it involves changing the interpreter which not only takes considerable effort (e.g Invisitype added around 2000 lines of RPython) but also implicates that with every new version of the interpreter come even further possible changes.

To address this problem, [17] uses a library written in python in which there are no changes to the compiler whatsoever. The structure of the library is based on decorators which are functions that can easily extend the behavior of other functions without modifying them. The library is limited to explicit flows.
since the tool acts under the assumption that the code was created by non-malicious programmers. It is also relatively easy to use, the programmer needs to mark the sources of untrusted input, sensitive sinks and sanitization functions and all that we are required to do to start using the tool is import it as a library. The downside is that the programmer must have the knowledge to recognize the functions and variables which he needs to mark and there is a need to access source code.

3.3.2 Symbolic Execution

Symbolic execution allows for exploring several execution paths at the same time without supplying the required input to follow them. An interpreter analyzes the program but instead of using inputs to travel down the code, it abstractly represents them as symbols. Consequently, the result is expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

This solution is normally more efficient than testing random inputs as the vulnerability may only be apparent for very specific input. Automatically exploring the space of possible inputs is vital. Theoretically, this approach would be perfect when done correctly. All possible vulnerabilities are detected and every input that is considered vulnerable is in fact vulnerable. But in reality it comes with a few challenges.

- **Path explosion**: This solution does not scale well with large programs since the number of possible paths will increase exponentially and will start impacting the performance of the program in terms of overhead. Not only the size of the program matters but language constructs like loops may exponentially increase the number of possible paths that have to be analyzed.

- **Environment interaction**: Problems can derive from the interaction of the program with components that are not under the tool’s control. For example system calls or accesses to libraries.

- **Constraint solving**: Constraint solvers are used to construct instances of the program that would cause property violations. Programs can scale into hundreds of variables which damages the technique in terms of efficiency.

In [18] the cost and complexity of full symbolic execution is avoided by using path constraints that mix concrete and symbolic values. These constraints are solved incrementally in a search for satisfying program inputs, lazily instantiating axiomatized models of executed functions as they are needed.

Three more tools based on dynamic symbolic execution are analyzed in [19] but even though some of the tools [20],[21],[22] solve some of the disadvantages mentioned above, they generate other ones.

3.3.3 Type checking

In order to make sure the types used in a program are correctly built and used in a valid way it is necessary to have a technique or tool that provides this guarantee. Most programming languages have a built-in type checker. Python has a strong dynamic type checker, which means that the types of
variables for example are checked as the program is running, during execution. But sometimes we want to make sure there are no run-time errors in the code, which would be the result if a type check flagged an error. Therefore it is useful to also statically analyze the types of the language. This will not only allow for an increase in performance since, like everything done statically, it is done before the program is executed. But will also guarantee that if the static type checker is well implemented, the program is executed with no errors due to type checking.

In the context of this project, we are only interested in python type checkers. It’s important to mention that the Python language already does type checking dynamically. On the other hand and due to the nature of our system, a static solution is also needed.

Its use relies on making annotations in the source code about the types of the variables and details of the functions used. It does not interfere with the dynamic typing of python and when the code is executed with the annotations mentioned before it does not stop its execution. This tool only provides suggestions before the program is executed but can be very useful since type errors are very common, especially in large projects with a lot of lines of code.

There are a few static analyzers available for Python which are open-source. Pyre[23] is one of them. Even though it does not support annotation through comments which is a considerable disadvantage, it allows for queries that return type-related information without the need to do a full type check to the program. Useful examples of queries can return a type of an expression in a certain line and column or if a type is a subtype of a specific class.

Contrarily to Pyre, Pytype [24] supports variable annotation through comments and the through the most used method. It has a very interesting feature that allows the type checker to disable a specific error and also provides type inference. The type inference can sometimes take a long time but if a file is taking too long to type check the tool offers the possibility of skipping it.

In the context of our problem type inference is not very useful since we will want to annotate it with a specific type that will remit you to the security level of that variable.

Besides the two type checkers referenced before, there is also PyCharm [25]. It is however locked to the IDE, which translates on the programmer having to install and use a specific IDE if he wants to use the type checker.

MYPY[26] is however different from PyCharm in the sense that it highlights the errors in a current file while MYPY shows errors in all files in your project through the terminal. This will allow us to type annotate third-party libraries for which we have the source to. Like Pytype it provides both annotation options and a stricter type checking than PyCharm’s. It is tunable by various flags and configuration settings and it is also not tied to an IDE which is specially advantageous when multiple individuals are working on the same project.

MYPY does not provide type inference but we already concluded its usefulness in this case. Because of its powerful and accurate analysis and the advantages mentioned earlier it was picked as the basis for this work.
3.3.4 Information Flow

Information flow is very useful, when verifying that there are no leaks between program variables that violate certain security requirements. Mainly due to its flexibility in terms of which properties it can ensure, the mechanisms it uses to maintain non-interference and its approaches to deal with changes in the variable’s levels, for example declassification or endorsement.

[27] performs a hybrid of static and dynamic control/data flow analysis. Static analysis analyzes implicit flows, flows that leak information through the program control flow, while dynamic analysis efficiently tracks dynamic information and determines definition-use pair. This pair reveals the difference between the definition of a variable and when it is used.

The main advantage of this hybrid approach is that it detects problems with ambiguous information via a dynamic check which avoids the need for approximate static analysis. Moreover, the information flow is inferred in the context of a low-level language named Python bytecode, which is the compiled python code, since source code is usually unavailable in real world applications, for which web security is a real issue.

Due to the fact that there is only one tool regarding information flow specifically for python and given the relevancy of the matter for our system we also look into information flow applied to full fledged programming languages in order to understand if something can be used.

3.4 Information Flow for full fledged languages

As was said earlier, this subsection will address information flow applied to other full fledged languages. Starting by Java, Paragon [28] is a language created on top of Java that allows for static checking of information flow policies. The policies are done through the use of modifiers. These are keywords that you add to those definitions to change their meanings (e.g Public or Private). The use of modifiers allows for the separate analysis of policies and types. Another important aspect of this tool is that it contemplates both explicit and implicit flows alike.

[29] Like Paragon, JFlow builds a language which is an extension of Java and uses formal rules to check if JFlow programs are correct. The verification of JFlow programs is mostly static which has the advantage of generating low run-time overhead and only uses dynamic analysis when the static one is too restrictive. The dynamic analysis itself is done to check if there is no leak of information by the success or failure of the run-time measure itself. The language also allows for means to deal with implicit flows, more specifically the compiler binds a program counter label to every expression or statement evaluated which can be used to track the information that might have been leaked from the statement or expression.

Jif is a security-typed programming language that is based on Jflow with support for information flow control and access control, enforced at both compile time and run time. Jif provides richer support for tying security requirements to programs, with important features like selective, robust downgrading,
language-based access control, and dynamic labels and principals.

LIO[30], an extension of Haskell, a purely functional programming language, uses a dynamic approach in which programs can encapsulate and return the results of computation with labels attached. The labels are attached in the form of a type, at run-time and the policies associated with those labels can ensure Non-Interference for integrity and confidentiality.

In [31] a library named Aglet embeds security-typed programming in a dependently-typed programming language (Agda). In this library information flow policies are used to restrict the use of results based on what went into computing them, e.g. tainting user input to avoid SQL injection attacks.

In [32] approach, the operating system identifies a set of input channels as illegitimate, and the processor tracks all information flows from those inputs. A broad range of attacks are effectively defeated by checking the use of the illegitimate values as instructions and pointers. During an execution, the processor tracks the information flows mentioned earlier. With the tracked information flows, the processor can easily detect dangerous uses of illegitimate values and trap to a software handler to check the use. For example, checking if an instruction or a branch target is illegal prevents changes of control flow by potentially malicious inputs and dynamic data generated from them.
Chapter 4

Model

As it was hinted in the former chapters our solution is based on information flow. In this chapter we describe the approach taken to ensure that the server side of the e-voting system tagus is secure. This chapter will not be focused on the tool's implementation since it will be covered in the following chapter but give you a general idea of the algorithm and insight on some of the choices made.

4.1 Information Flow

Prior to jumping straight into this tool's workflow let us recall one very important decision, why we opted into static analysis. The most important factor was the context in which this analysis would be performed. Since the target of our system is an e-voting system, its availability is of a critical nature. As soon as the election starts it cannot be stopped midway which makes dynamic analysis unviable. If a vulnerability was discovered it would not be possible to stop the process.

The basis of this tool is information flow. In order to achieve it we would need to allow information to flow in one direction but issue some kind of warning when flowed in the contrary direction. This was done resorting to inheritance. Inheritance is a very useful mechanism where classes inherit properties and behaviors from their superclass. Thus class would be called a subclass. Its main use lies on the reusability provided by it since there is no need to redefine the already present functionality that it inherited from its superclasses.

However useful that property may be, it isn’t in it that we are interested in. What makes inheritance so useful for this approach is the simple fact that it allows information in one direction but not the other. This will be exemplified through the most basic form of information flow, an assignment. Its sole purpose is to allow the information to flow from one variable to another.

4.2 Sub-class Structure

Generally, an assignment between two variables may only be performed if both variables hold the same type. However, what would happen if you tried to create an assignment between a variable with the type
of the superclass and one with the type of the subclass? A sub-class object is also a super-class object. So one can assign a sub-class object to a super-class variable. The opposite is not secure, since a sub-class can define additional behaviour.

But, as we can see in figure 4.1 because the subclass can have additional subclass-only functionalities, assigning a superclass reference to a subclass variable is not allowed and will result in a type error. Let us now assume that the superclass is equivalent to a secret variable and the subclass to a public knowledge variable. If we try to create an assignment in which the superclass is assigned to a subclass we will be in the presence of an illegal information flow.

![Figure 4.1: Inheritance.](image)

It is also worth mentioning that downgrading the level of a specific security property is also possible using this approach. We have already seen the reasons for why downgrading is vital in an information flow context in section 2.3. To be able to apply either endorsement or declassification to a variable we have only to change its type.

### 4.3 Security Lattice

Since we have now established a way for us to identify illegal flows between two classes we can leverage that along with the concept of multiple inheritance to be able to extend that into multiple security levels and the corresponding relationships between them. We will be able to create a lattice like the one observed in figure 4.2, that represents the possible security levels with inheritance connecting the different levels.

The security levels previously mentioned will be related to two different security properties. These are composed of integrity which assures that the information is accurate and trustworthy, and confidentiality that limits the access to information. The two properties work inversely to each other which means the
highest level in the lattice in terms of confidentiality will be equivalent to the lowest level in integrity.

### 4.4 Explicit flow

We now have a mechanism that allows us to compare two variables whose types belong to an inheritance hierarchy and check that there are no illegal flows between them. However, without a procedure that allows us to attribute the corresponding security levels to variables the just described mechanism would be rendered useless. Since the security levels are nothing more than types, we would need a convenient way for the programmer to attribute types to variables in a static environment. The solution lies in type annotations. At first, the concept of type annotations in a dynamically typed language like Python may raise some questions. Even though Python only checks the type of the value of a variable during run-time, it may still be beneficial to have a hint of what that type might be to facilitate the comprehension of the code.

In the context of our problem, the type annotations serve as a bridge between the security level lattice and the variables relevant for the programmer. As a result of the type annotation we can effectively change the types of the variables so that a type checker could identify when there would exist type errors. Since the types of the variables are now security levels, each type error would be equivalent to either a confidentiality or integrity leak.

The type annotation itself is fairly easy to perform as represented in listing 4.1. The complexity comes from the programmer having the necessary knowledge to know which level to attribute to each variable. That adds flexibility to the programmer in the sense that he can decide the various scenarios in which a variable may have different security levels but also adds a great responsibility. The programmer’s type annotations directly decide the tools’ efficiency.

Type annotations can be added to variables in assignments as illustrated in figure 4.1, to arguments of functions and can even represent the return values of said functions.

```python
// Type annotation in assignment
Var = "test" # type: HighConfidentiality
```
As we could see in listing 4.1 we have laid the foundation for our tool to work but still lack an essential step, the type checker. The type checker, more specifically MYPY, will check the code for type errors and since the types are now security levels, it will output a warning when an illegal flow is discovered. The reasons to why this type checker was selected among the few that exist are described in the background chapter. Initially the chosen solution was to change the type checker MYPY to be able to perform information flow without resorting to any external interaction. Although promising, this solution proved to involve too many changes in too many places due to the gigantic size of the type checker.

4.5 Implicit flow

If all the security leaks originated from assignments, MYPY would actually be equipped to deal with them, aided by the security lattice. This analysis is, however, insufficient since there are many more instances where security leaks can arise. A clear example of such an instance would be a leak through an implicit flow. Before we take on the challenges of implicit flows it is important to consider the context of the program this tool will be applied to. Since the main focus of this tool aims to make the target code as secure as possible, the approach will be as conservative as possible. As so, it will eventually risk some false positive results in order to avoid false negative ones. We already mentioned the existence of implicit flows but it is also important to understand how they occur in the context of source code. These instances where implicit flows occur include the presence of a conditional statement or a loop.

Resorting to figure 4.3 let us consider the conditional statement where different computations can be performed based on the conditional expression. To better understand it, imagine the following scenario: a person is trying to figure out if a sandbox has been used by a cat. Yet that person only has access to the information lying on the residing sand. Depending on the state of the sand, for example if it has footprints imprinted on the sand, one could deduce if a cat had been present without ever actually seeing it.

Going back to our example, if both the possible computations contained observable variables, like the sand, and they differed in value, like footsteps in the sand. It would be possible to obtain information about the conditional expression itself without having direct access to it. If the conditional expression contained confidential data, we would be in the presence of a security leak. Depending on which branch was computed, it would be reflected in a change of the observable variable and we would gain knowledge about the conditional expression.

The thought process of the occurrence of a security leak in a loop would be similar to the just described scenario of the conditional statement with the following difference. Instead of observing changes
in the variables depending on which computation was handled, the attacker may observe changes depending on how many times the cycle is performed. As in the former example, if the body of the loop contains an observable variable, depending on how that variable changes information of the condition itself can be inferred. If the condition contains confidential information, we are again in the presence of a security leak.

4.6 Built-in functions

Another example of a security leak would be in the presence of built-in functions. The most famous example would be the appearance of an `eval` function that allows the evaluation of a specified string as code followed by its execution. It is common knowledge that allowing raw input to be executable without the slightest analysis is at best reckless. But the security risk isn’t limited to the `eval` built-in function. In fact most of built-in functions can be considered a liability in certain conditions, for example the use of the built-in function `print` with its content being a confidential variable would disclose the information.

It would be hard to argue that these constructs are not vital for programmers as they are present in most pieces of code. To address this the approach taken would involve translating them into a state that allowed `MYPY` to analyze them. Since `MYPY` can analyze illegal flows in assignments, in practical terms, it would mean adding several assignments that would be equivalent to analyzing the constructs themselves.
4.7 Coherence regarding changes

Navigation through the code will be done resorting to the Abstract Syntax Tree which will be, throughout this paper, mentioned as AST. The AST is a tree representation of the abstract syntactic structure of the source code. It is used by the Python interpreter to help generate code that can be executed. Each node of the AST represents a basic element of the code. The figure 4.4 is a graphical representation of the AST. In this example you have the representation of a function call to \texttt{function} with the argument 10 and 11.

![Figure 4.4: AST representation](image)

Being able to navigate through the code will make us able to identify the extraordinary cases and add the assignments in the correct places.

However, changes to the source code come with several challenges. The most important one would be the correspondence of lines between the source code and the now changed code. The programmer will receive warnings when illegal flows arise, and these will have to include their location. This topic will be explained in depth in the next chapters but we can already identify a few challenges that come with this approach. Naturally, there may be no repercussions to the programmer in regard to the added complexity of this approach. Since that translates into the programmer not having access to the changed code, in order for the warnings to be accurate in terms of their location, the correlation between the changed code and the source code must be done correctly.

Having successfully examined the code and identified the security leaks, the output given by \texttt{MYPY} would still need some adaptation. Given that its purpose is to detect type errors, at this state, the output of \texttt{MYPY} running through our changed program would be the multiple type errors between different security levels as can be observed in listing 4.2.

```python
1: Incompatible types in assignment
   (expression has type "HCLI", variable has type "LCHI")
```

Listing 4.2: Native MYPY output

Since the purpose of the output is now informing a programmer of the possible illegal information flows the output could be improved upon. For each warning it would contain the content of the source code, the variable that originated the warning and obviously the line in which the warning occurred.
Chapter 5

Implementation

This section describes the implementation thoroughly. It starts by an explanation on what is the general workflow used and what is the purpose of each step of the algorithm. Following the same structure, this section then dives further into how each step of the algorithm is being performed. The thought process of the algorithm behind the developed tool, as well as a brief explanation on the most complicated challenges and the corresponding solutions to address them.

It is concluded with a documentation on what were the achieved guarantees and limitations of the tool. All of its implementation will be available at the following github page [33].

5.1 Workflow

The workflow of the developed tool consists mainly of four stages as presented in figure 5.1. The purpose of each of these stages will be explained in the following sections, but it is important to understand that only a broad view of each step is going to be given in this section, leaving the implementation
5.1.1 First stage - Initial Transformation

The initial transformation involves changing the python code that is going to be subject of the security analysis. However, prior to the execution of this step the programmer will already have performed the following measures. He will already have added the type annotations to the variables.

The annotations are equivalent to security levels of the variables and will be used by the tool to compare them. The programmer is responsible for adding the type annotations so he must have prior knowledge on which variables require security levels and which level is attributed regarding confidentiality and integrity. This step is the only one that requires the programmer's attention and it is essential that it is done correctly in order to get an accurate analysis of the target code.

Apart from adding the type annotations the programmer is also required to provide the tool with his security lattice. Since there are two security properties, confidentiality and integrity, the tool will require two lists with the multiple security levels that the programmer wishes to have at his disposal. Once the lists are provided, the tool will be responsible for handling the relationship between the variables in them.

As soon as these two prerequisites are dealt with, using the original code provided by the programmer as input, the tool will make sure it only has vital components to be analyzed. More specifically it gets rid of useless information which are, in the tool's perspective, empty lines and all of the comments done by the programmer. The exception being the comments related to the type annotations mentioned earlier. This step converts the original program into a ready to analyze type annotated program represented in the figure 5.1 as an R program.

5.1.2 Second stage - Type Annotation

The second stage comprises of changing the variables' type into the type defined by the annotation. As it was mentioned before, we will resort to the AST to navigate through the code until we reach a relevant assignment. As the search in the AST progresses and the assignments are being found, if in the presence of a valid type annotation, the tool will perform its next operation. It will convert the right part of the assignment into the object of the type annotation. As shown in the listing 5.2 the value of the variable is converted to the security type annotation which, for simplicity, only takes confidentiality into consideration.

```r
// before the conversion  // after the conversion
1: testVariable = "anyvalue" 1: testVariable = HighConfidentiality
# type: HighConfidentiality
Listing 5.1: Variable original state

Listing 5.2: Variable transformed
```

To generate a ready to be analyzed program that includes the Security level annotations, R + Security level annotations, the variables would have to effectively belong to the type defined by the programmer,
which again translates into its security level.

We will now dive into how we deal with operations. We consider operations to be functions that are equivalent to the built-in operators of Python. An example of maybe the most basic operation is the `add` function which takes two variables and performs the mathematical computation equivalent to a sum. In this context the two variables are called the operands. Note that operations are not limited to mathematical computations, another common example of an operation would be the equality that makes use of the operator `==` and exists to check if two variables hold the same value.

Through this example we can already realize the importance of dealing with operators. It would be interesting for us to be able to discern if an operation would output a potentially dangerous result. Using listing 5.3 as an example, if weren’t able to deal with operations we would not be able to spot the illegal flow that would occur in line 4.

Using a conservative approach, if just one of the variables is of a higher security level, in our example `var1`, the result from the operation should also be of that same higher security level regardless of the other variable's security level. We can clearly observe that the result from the operation in the right part of the assignment in line 4 would be of a High security level and since the left part of the assignment has a lower security type we would be in the presence of a leak. Regarding the security related types, `HCLI` stands for High Confidentiality and Low Integrity which will be referred to as the acronym for simplicity purposes.

```
1: var1 = HCLI
2: var2 = LCHI
3: var3 = LCHI
4: var2 = var1 + var2
```

Listing 5.3: Operation scenario

As to the actual approach, like represented in listing 5.5, it involves the replacement of the operation by a call to a specific function, `operationHandler`. Through that function we can compare the security types of `var1` and `var3`. How exactly will be explained further on, when we dive into the implementation details in the next section. Before this approach could be performed, the generic class `A` as well as the `operationHandler` function will be created and inserted at the start of the target code.

```
// before the changes
1: var1 = HCLI
2: var2 = LCHI
3: var3 = LCHI
4: var2 = var1 + var3

Listing 5.4: Operation original state
```

```
// after the changes
1: var1 = HCLI
2: var2 = LCHI
3: var3 = LCHI
4: var2 = operationHandler(var1, var3)

Listing 5.5: Operation transformed
```

This approach is also used when providing downgrading. Let us assume that after a certain point a
variable no longer holds a high confidentiality value, to avoid false positive warnings the user may de-
cide to apply declassification, considering confidentiality, or endorsement, if he wants to target integrity.
Taking listing 5.7 as reference, in line 3, a warning will be output. Yet in line 5, the same assignment will
no longer result in an output since we applied both declassification as endorsement to it.

```
// before the changes
1: var1 = HCLI
2: var2 = LCHI
3: var2 = var1
4: var1 = ""
# type: declassification (LCHI)
5: var2 = var1

// after the changes
1: var1 = HCLI
2: var2 = LCHI
3: var2 = var1
4: var1 = LCHI
5: var2 = var1
```

At the end of this stage the code will no longer be executable and will lose some information regarding
the values of variables. As you can see in listing 5.2 we will no longer know that testVariable held the
value anyvalue. The values of the variables are no longer important for us. What is of the utmost
importance is detecting illegal flows between variables, and to do so we only need the variables to have
a security-related type. It is also important to note that the programmer will always keep the original
code.

### 5.1.3 Third stage - Assignment Creation

The third step is possibly the most complicated one. As we have seen in chapter 4, there are multiple
scenarios that wouldn’t be addressed if our analysis were to stop at this point. For explanation purposes
this step will be divided into two but both of them are happening simultaneously. The first involves the
creation and addition of assignment expressions which we will represent as \( R' \). This will be done in
order to deal with the missing scenarios that have been described in the former chapter and will again
be referred subsequently. The second is a complimentary method whose purpose is connecting the
original code to the increasingly different code that we are changing at each step of the workflow.

**Generating \( R' \)**

After the second stage is finished, the following step involves the analysis of not only built-in func-
tions of the language but loops and conditional statements as well. Built-in functions are, as the name
implies, functions that are built into the language and whose functionality is predefined by Python. Ex-
amples of such functions would be `print` or `eval`. The goal is to make MYPY able to process those
functions. To be able to do that, the arguments of the built-in functions are converted into assignments.
As we can see in the listing 5.9 two extra assignments were created and inserted in lines 3 and 4 of this snippet of code. In this specific case the built-in function was print. Just by looking at its definition which basically allows for a specified message to be printed to an output device, we can see how this function can cause a confidentiality leak. Resorting to the example, all it would take was for Variable to have a high confidentiality level, represented here by HC. The rationale behind the added assignments is to be able to check if Variable is indeed highly confidential. In line 4 we have exactly that. Since in line 3 we have defined leftside1 as being the lowest security type possible (LC), if Variable is anything but the lowest security type, MYPY will generate a type error. By creating two new assignments we can now successfully analyze built-in functions.

With the built-in functions dealt with, we still have to address loops and conditional statements. Since the thought process is similar for those two scenarios we will just take the loop construct example and explain the conditional statement more in depth in the next section. We remind ourselves that we have already narrowed an implicit flow in the context of a loop so a single scenario. An implicit flow can only exist in a loop if the condition contains a High security level variable and the body of the loop contains a low level one.

As we can see in listing 5.11 an assignment was created and inserted in line 4 of this snippet. If VariableList is highly confidential and inVar is of a low confidential level, MYPY will be able to generate a warning and we will be able to address implicit flows in loops. The current example is a for loop but naturally the rationale can be extended to while loops as well.

Log of Changes

While the R’ state is being generated, the correspondence between the lines before and after the changes to the code is also being recorded. This is being done so that the location of the warning fed to the programmer, which will be part of the end result of this tool, is accurate. The location of the
warning is given by its line number. Resorting to the listing 5.13 below, if MYPY ran through the code after the changes it would output warning messages in both lines 5 and 8. In the original code, the warning messages would be related to lines 3 and 4.

```
// before the changes
1: Variable = HC
2: Variable2 = HC
3: print(Variable)
4: print(Variable2)
```

Listing 5.12: Original code

```
// after the changes
1: Variable = HC
2: Variable2 = HC
3: print(Variable)
4: leftside1 = LC
5: leftside1 = Variable
6: print(Variable2)
7: leftside2 = LC
8: leftside2 = Variable2
```

Listing 5.13: Transformed code

At first this may not look like a critical problem but considering that programs can be composed of thousands of lines comprising of hundreds of variables with hundreds of function calls, it would be impossible for the programmer to identify the source of the warnings.

In light of this discovery we are also in need of a log that allows us to convert the line of a certain warning in the current state of the code into the original code. We will call it log of changes from now on. At this stage we have achieved the two requirements to apply the final step, a complete log of changes, and the R' state program.

5.1.4 Fourth stage - Warning Creation

The final step involves running MYPY through the R' state program and resorting to the log of changes, adapting the warnings provided by MYPY. This adaptation involves mainly the correspondence between the lines of the original code and the R' state code. As soon as this process is finished, the end result will come in the form of warnings. These warnings, as you can see in listing 5.14, include all of the relevant information required for the programmer. It provides the line in which the leak has occurred as long as the specific line of code.

There is an error in line: 26
The corresponding code is: public = secret

Listing 5.14: Warning example
5.2 Implementation Details

This section will go into detail about the implementation of the tool and explain some of the options that were made. It will follow the same structure as the workflow yet instead of focusing on what was being done, it will focus on how it was being done.

5.2.1 First Stage - Initial Transformation

Like mentioned in the last section, the initial transformation’s purpose involves the cleanse, in the tool’s perspective, of the original code. It implies getting rid of empty lines that are not necessary and every comment done by the programmer that is not related to the security annotations. What wasn’t mentioned before is what exactly is meant by creating what is the equivalent of a security lattice. The programmer will provide the two lists as said before, organized by their security levels and them being higher as we go through the list.

Once the programmer provides us with the security lists we will have to generate a class for each of the levels. Let us assume the list only holds two security levels, high and low. Which means there will be high and low confidentiality levels (HC and LC) as well as high and low integrity levels (HI and LI).

As we saw in the chapter 4, in figure 4.2 we can see the resulting lattice from the two security lists.

```python
1: class HCLI () :
2:     pass
3: class HCHI(HCLI) :
4:     pass
5: class LCLI(HCLI) :
6:     pass
7: class LCHI(LCLI, HCHI) :
8:     pass

Listing 5.15: Class Inheritance Hierarchy
```

In the listing 5.15 we can see the classes that were created in order to address the security requirements given by the lattice. In line 1 we can see the highest level confidentiality class which has no superclass and in line 7 we can see the lowest confidentiality class. Note that even though the highest and lowest classes are not directly connected they can still be compared. The classes do not require methods or arguments since their sole purpose is being the foundation for our inheritance-based structure.

In terms of implementation there is nothing special about the first stage itself as its only step involves the contents of the original code being passed to a file. The transformation performed in listing 5.17 prepares the code for the analysis. What results from this transformation will be what we call the original code. This step is performed in order to have a consistent starting point in terms of the state of the code. The tool requires this transformation to be able to accurately generate the log of changes. This code will
be given to the programmer as its original code.

```python
// before the changes
1:
2: def exampleFunction():
    # This comment will be deleted.
3:
4:     var = "testString"
    # type: HCLI()
5:     print(var)
    # Prints the variable.
   Listing 5.16: Original code

// after the changes
1:
2: def exampleFunction():
3:     var = "testString"
    # type: HCLI()
4:     print("Testing")
   Listing 5.17: New Original code
```

### 5.2.2 Second Stage - Type Annotation

The second stage involved traversing the AST and converting every variable that has a type annotation in the form of a comment into the variable of that type. This is done by identifying the initialization of the variable, which is normally an assignment and replacing the right part with the corresponding type regarding its security level. In this listing 5.19 we can see the two variables being modified as stated before.

```python
// original code
1: variable1 = "example"
    # type: LCHI()
2: variable2 = 20
    # type: LCHI()
   Listing 5.18: Variable original state

// after the changes
1: variable1 = LCHI
2: variable2 = LCHI
   Listing 5.19: Variable transformed
```

Implementation-wise we are allowed to do this due to the existence of a parameter called `type_comment` that exists in every assignment node and will survive the cleansing performed in Stage 1. As we are traversing the AST besides identifying the assignment of the variable we also have to check in the parameter has a valid type annotation, which in our context will mean that it belongs to the inheritance structure that we mentioned before.

In the following example 5.20 we observe the representation of a node of the AST, note the last field, `type_comment` that allows a type annotation to be made. If a comment in the code starts by `type: Python` adds the type to that field.

```python
ast.Module(
    body=[
```
Listing 5.20: AST Node Representation

Like exemplified in the listing 5.21, we compare operands by applying them the Union function. This function is a MYPY feature that allows for the grouping of multiple type values. This will allow us to compare the type of the left side of an assignment with the multiple types of the various operands. This will translate into an illegal flow if just one of the operands is in an illegal situation, which is consistent with our conservative approach.

```python
1: def operationHandler(x : S, y : T) -> Union[S, T]:
2:     return

Listing 5.21: Operation Function
```

In listing 5.21 we can also see that the security types will be the given to the function as type annotation of the arguments. As we can see from the output, the function will essentially try to unite the security types. As so it will generate a variable with the highest type of the two arguments. In this case it will compare S and T.

### 5.2.3 Third Stage - Assignment Creation

In this stage we will go deeper into the approaches to deal with both built-in functions and implicit flows. We know from the previous chapter that the solution is based on creating and adding assignments.

**Built-in functions**

The remaining leaks related to the explicit flows would come from what we call built-in functions. These functions are built into the Python interpreter and are always available. Some of them are harmless while others pose a serious threat to the security of a program if not handled correctly. An example of a well used built-in function would be `print`. MYPY itself cannot analyze the outcome of built-in functions in terms of leaks, even if the types of the variables used in said functions represent the security levels.
Since MYPY can only analyze the most simple leaks related to explicit flows, assignment derived leaks more specifically, the following approach was used.

While traversing the AST when the tool finds a node corresponding to a built-in function, it adds two assignments in the following lines. The first assignment is the creation of a dummy variable with the type corresponding to the lowest level of both integrity and confidentiality.

```python
// before the changes
1: Variable = HCLI
2: print(Variable)
   
Listing 5.22: Built-in original state

// after the changes
1: Variable = HCLI
2: print(Variable)
3: leftside1 = LCHI
4: leftside1 = Variable
   
Listing 5.23: Built-in Transformed
```

As you can observe in listing 5.23 the dummy variable in line 3 always has the name leftside followed a number so we know how many built-in functions were found before the occurrence in question. The right side is always LCHI because of the reason stated earlier. The second assignment is the most important one because it is in that line that MYPY will detect a leak and trigger a warning. The second assignment's left part is the dummy variable that we created in the first added assignment while the right part of the assignment will correspond to the variable inside the built-in function.

Note that if we have more than one variable inside the built-in function we will create two lines per variable so that we can analyze each variable by itself.

```python
1: a = LCHI
2: b = LCHI
3: c = HCLI
4: print(a,b,c)
5: leftside1 = LCHI
6: leftside1 = a
7: leftside2 = LCHI
8: leftside2 = b
9: leftside3 = LCHI
10: leftside3 = c
   
Listing 5.24: Multiple variable Built-in
```

For example in listing 5.24 even though there are 3 variables in one statement we can pinpoint the origin of the leak in terms of which variable caused it. We know it was variable c that triggered the warning by checking in which line was the warning triggered, in this case line 10.

This is another example which makes the correspondence of lines between the original code and the current state of the code so important for this tool. Since the warning lines will be totally different from
what the programmer observes it is essential to have a way to correlate the two. As we can see in listing 5.24 the warning line number which for this example would be 10, would be in a different line (4), in the original code, which is the only thing the programmer has access to.

Consequently the tool is also saving the line in the original code and the contents of that line in a dictionary-like structure so that the bridge between the original code and the code with the added lines can be done. This structure was chosen because given the warning line, it can easily be used as a key to access the dictionary which will return the corresponding line in the original code and the content of that line. This structure will be further explained as we progress this section.

We also want the programmer to be at ease to use built-in functions if the arguments are built-in types that aren’t related to security like the string shown in the following listing 5.25.

```
1: print("testing")
```

Listing 5.25: Harmless built-in

In this case, line 1, there would never be an illegal information flow so it doesn’t make sense to add the extra assignment since MYPY doesn’t need to check the function’s argument in that case.

**Conditional constructs**

We can now handle explicit flows such as assignments and built-in functions. All there is left to cover is how to deal with illegal implicit flows.

With respect to implicit flows, we will only be addressing three possibilities of information leaking. In reality, there are more possibilities, some are external to the code like computational time and are, therefore, outside of the scope of this work but others make use of built-in constructs to generate these flows.

One quick example would be exceptions. The fact that an exception is received by the attacker may give information regarding variables for which he shouldn’t be allowed to access. Let us image that there is a confidential object for which the attacker has no permissions to access. That object may have certain fields and methods that the attacker should not know exist. While resorting to the listing 5.26, let us image that he uses the function in line 7 while looping its second argument which would be the equivalent of testing continuously if the field existed in the object. If it does not exist it will output an exception `AttributeError` and if it does it won’t output nothing at all. Note that even though he has no access to the object Secret, he is able to know that it has a `Location` and a `Name`. 
```python
class Secret:
    Location = ""
    Name = ""

a = Secret()
getattr(a, field)
Listing 5.26: Implicit flow through built-in function
```

However, we are only addressing the three possibilities that are linked to conditional statements and loops. In these three cases information leaks can be reduced to one scenario. If there is a high level variable present in the control statement of a loop, and a low level variable present in the body, we are in the presence of a leak. In listing 5.27 there is an example of the latter statement where secretVar is a high level variable and publicVar is a low one, if we consider only confidentiality. Depending on the value of b that we as an attacker observe we will have knowledge from the state of a. However, in terms of implementation the subject is not that simple. Since we operate in the static realm, we don’t know which direction the program will take since we don’t have access to the value of secretVar in this case. Given that our approach is always conservative we will consider both possible computations and issue a warning even if only one of them can actually originate one. As we can see in listing 5.28, after the changes, we have added two assignments (lines 6 and 7) which will address and identify illegal flows in both branches if they exist.

```python
// before the changes
1: secretVar = HCLI
2: if (secretVar):
3:     publicVar = 1
4: else:
5:     publicVar2 = 2
Listing 5.27: Conditional construct original state

// after the changes
1: secretVar = HCLI
2: if (secretVar):
3:     publicVar = 1
4: else:
5:     publicVar2 = 2
6: publicVar = secretVar
7: publicVar2 = secretVar
Listing 5.28: Conditional construct transformed
```

In the presence of a conditional construct all the variables inside both possible computations are gathered, filtered for duplicates, and compared to the variables inside the expression. Once we have gathered all the relevant variables we find out where the conditional construct ends and insert the assignments after.
**Loop constructs**

Before we address the implementation details behind loops let us first briefly define them. A loop-like structure will be divided in two. As we can see in listing 15.29, the control statement, which is where the condition that direct its body lies (line 2), and the body itself (lines 3, 4).

```python
1:
2: for var in List:
3:     counter += var
4:     pizza += 1
Listing 5.29: Loop construct example
```

Like always the aim is to change the AST for MYPY to be able to directly analyze the program, which is simpler terms involves the conversion of the information we have into assignments. In this case we must make sure that if the control statement has a high security level variable and the body contains a low one MYPY triggers a warning.

In order to achieve it, we would again analyze the body of the loop and retrieve all the variables in it. Which in the listing 5.30 would result in a, b and c.

```python
1:
2: for str in strList:
3:     a += 2
4:     b = b + a
5:     c = "exp"
6:
Listing 5.30: Loop construct
```

In a second stage we filter those variables, for example if the variable is inside a built-in function, it is already handled so there is no reason for us to analyze it again, or if the variable is both inside the control statement and inside the loop figure. Turning our attention to listing 5.31, we can see the useful variables in the lines 3, 4 and 5 that represent the variables of interest and the variables in line 6 and 7 that will be ignored since they were already either in the control statement or inside a built-in function.
Finally we find out where the loop ends and construct and insert the assignments after. The assignments are represented in listing 5.32. In order to test all the options we would have to compare all the variables inside the condition statement with the ones present in the loop's body. In this example there is one variable inside the condition statement and three useful ones present in the body. This results in three assignments that we can observe in lines 10, 11 and 12.

1: for (str in strList):
2:     a += 2
3:     b = b + a
4:     c = "exp"
5:     print (d)
6:     str += 1

Listing 5.31: Loop construct containing built-in function

Listing 5.32: Loop construct transformed

5.2.4 Fourth Stage - Warning Creation

In this final step, MYPY is ran through the changed code which will generate both the useful warnings in the wrong lines for the programmer and warnings that are not useful for this tool. In listing 5.33, there are a few examples of warnings that can appear that are not useful for the programmer is this case. There are a lot of reasons for why these warnings appear, for example if there are variables without any type hint attributed to them or MYPY expecting an argument for a function that is missing.
Example1: Need **type annotation** for "variable".

Example2: Too few arguments for "function" function.

Listing 5.33: "Useless" warnings

These type of warnings are filtered by checking MYPY’s output to see if they have specific keywords in order to avoid confusing the programmer with unrelated warnings. Even though they may be useful they are not related to security and are therefore out of the scope of this work.

Even though we filtered the unrelated warnings MYPY’s output may still be confusing for the programmer. Not only are they in the wrong lines since the correlation wasn’t performed yet, but the warnings themselves only state a type error between variables. The observable warning in listing 5.34 would correlate into an illegal flow, both from confidentiality and integrity perspectives.

```
1: Incompatible types in assignment
   (expression has type "HCLI", variable has type "LCHI")
```

Listing 5.34: Original state warnings

As stated before, each time an assignment is added to the code through the AST, the useful information is saved in a dictionary structure. Each entry in the dictionary is composed by the line in the changed code, which is the key of the dictionary, the corresponding line in the original code, and the code itself that belongs to the original line. All of the information regarding the original file is stored as a value in the dictionary.

```
{50 : [43, print(f), True]}
```

![Dictionary cell.](image)

In some cases the structure has an extra field like exemplified in figure 5.2, a Boolean which is used to differentiate the entries that were originated from a built-in function and the ones that came from loops. We will get back to why those two cases require differentiation soon.
If the warnings only resulted from built-in functions or loops, we would only have to access the dictionary by its key and show the programmer the corresponding value in the dictionary which would be the original line and its content. But a warning can come from a simpler scenario, an assignment itself. As we can observe in listing 5.35, more specifically line 7, will originate an illegal flow.

20: secret = HCLI
21: public = LCHI
22: for (file in Filesystem):
23:     a += 2
24:     b = b + a
25:     c = "exp"
26: secret = public
27: for (numb in PhoneList):
28:     a += 2
29:     b = b + a
30:     c = "exp"

Listing 5.35: Explicit flow amidst loops constructs

Consequently, direct access to the dictionary will no longer be sufficient to identify all the relevant warnings. After checking the dictionary for direct correspondence, if not found, the tool will check how many lines were added before the line in question. It will do so by counting the entries in dictionary that is ordered by the warnings' original line, before reaching the warning in question. For the example showed in figure 5.3, assuming we would be in the presence of an illegal flow originated from an explicit flow in line 60. We would check the dictionary for how many entries it would have before line 60, and subtract the lines in order to get the original line that we could then feed to the programmer.

Since we deal with built-in functions by we adding two lines to the code and only one for each variable inside the loops cases, we would need an extra field to be able to differentiate the two entries. That is where the Boolean comes in, so that when the tool is handling a warning that isn't in the dictionary it can accurately count as many lines were added until that point.

In listing 5.36 we have what the final output of the tool looks like for the example mentioned earlier. It includes the line of the warning and its content as well as the variable that generated it. If there are multiple information leaks, the programmer will receive multiple warnings like the one we can see below. This specific one involves an illegal from from the variable secret to the public one in line 26.

There is an error in line: 26
The corresponding code is: public = secret

Listing 5.36: Final warning output
Warning Line: 60

Original Line: 60 – 6 = 54

c += 2

c += 1

c += 1

c += 2

c = 6

Keys

33
37
40
53
63

Values

[31, print(f), True]

[26, b=b+1]

[27, c=a+c]

[42, print(a), True]

[38, Var=c+1]

Figure 5.3: Dictionary.
Chapter 6

Validation

This section will involve two different sections, we will first dive into how each component and phase of the workflow was tested followed by the final evaluation through the use of the e-voting test case.

Based on the most common and used guidelines when testing a system that fits this description, the testing will be divided into two categories. It will be divided into unit testing, where every isolated component of the tool will be thoroughly tested independently and integration testing, which is when we will evaluate how all the pieces of our tool work when they have to depend on each other. Naturally it will be easier to catch simple errors when testing each unit by itself which will also make the integration testing easier since we won’t have to worry about one of the pieces not working and can focus on the relationships between components.

6.1 Unit Testing

Unit tests consist in isolating the developed code in logical units and then exhaustively testing them in isolation. Considering that target the code is likely to contain errors, these type of tests tests will be ran with each version of the unit to make sure the end result remains the same. Testing the possible inputs to a component is important, since it is not feasible or practical once it is integrated into the system. Unit testing may also help identifying issues in the implementation in general so it is important that is done as early as possible.

As mentioned previously this part of the validation will ensure that each component works correctly on its own since it isn’t practical once the unit is integrated into the system. The tool was divided in the following components:

- Initial Transformation: where the original code will be converted into the initial code that is fit to be handled by the tool.

- Expression analysis: every expression is replaced by the function with its arguments being the variables that composed the former expression.
• Explicit flows (Assignments): testing if the classes that regard the security levels and the direct interactions between them work correctly, which means the inheritance hierarchy is also correct.

• Built-in functions derived: for each of the built-in functions in Python making sure the tool behaves accordingly and all of the warnings are triggered on the relevant situations.

• Implicit flows: every instance where an invalid flow can come from regarding implicit flows. These flows were divided into three cases according to the presence of while loops, of for loops and conditional statement if/else.

6.2 Integration Testing

While unit tests validate each unit, integration testing combines the former individual components to test the system as a whole. The tool was exposed to inumerous pieces of code ensuring it was exposed to as many different scenarios as possible. The ultimate test would culminate in applying the tool to the server side of the e-voting system tagus which we will analyze in the next section.

6.3 Test case

As the final test, we expose our monitor to the server-side code of the e-voting system tagus. To make sure every step of this test is understood, there is some background that must be covered first.

6.3.1 Relevant background

Regarding the starting point, in terms of security lattice, we will use the lattice already present in figure 4.2. That will translate into us having two security levels for confidentiality and two for integrity. The levels are High and Low but as usual we will use HC when talking about high confidentiality, LC regarding low confidentiality. Analogously, the same rationale will be used for integrity.

It is also necessary to address certain details about the framework used to create the e-voting system tagus. This was the Django framework which was chosen [2] because it allows the creation of web application in an easy and fast way and is written in Python. It provides many modules which automatically take care of some troubles when dealing with web applications, like authentication, sessions, injection among others.

Additionally, Django also provides a built-in authentication system in its default configuration. In reality what Django provides is not only authentication, which verifies if a user is who they claim to be, but also authorization, that determines what an authenticated user is allowed to do. This is of our concern because most functions present in the tagus server have a decorator with the name login_required.

More specifically, the purpose of this decorator is to check whether a user is currently logged in. If so the function will be executed. This is a positive indication, however, it does not automatically exclude
that function from illegal information flows. In this context, the attacker may have obtained someone’s
credentials or even have valid credentials of his own. As such, emphasizing this as a positive sign, we
will act as if the decorator does not exist.

Along the built-in authentication system, Django also provides means for communication between
the server and the client. It does so by using the protocol HTTP. We are not going to dive too much into
HTTP but what must be understood is a very high level notion of how the communication is done.

Typically it is done in three phases:

• The client sends an HTTP request to the server.
• Django parses the request, extracts a URL, and then matches it to a view.
• The view processes the request and returns an HTTP response to the client.

This is relevant for us because the functions contained in the code that we are going to analyze will
always have a request as their argument which in practical terms is a client request to the server.

6.3.2 Test

Before we dive into the illegal information flows contained in the tagus server code, there is still one
missing step, perhaps the most important one when testing this tool. The type annotations are yet to be
done. They are of the utmost importance because if not well done, the tool will lose a lot of its potential
when finding illegal flows.

Having this in consideration, every type annotation will be explained in regard to why that security
level was used. It is also important to know that since some of them have the same rationale, we will be
genralizing the type annotation for certain scenarios not to risk repeating the same information.

Type annotations

The most general type annotation that we will be making is to arguments of the functions. Like mentioned
in the former section, due to the way communication is handled the request that the server receives in
each function will be an HTTP message from the client. And in the worst-case scenario, the input from
the user should always be regarded as a security risk.

Following that line of thought, as we can see in listing 6.1. For every function that receives a request
as argument, we attribute its argument the type LCHI. This type is the lowest security level that we
possess regarding both confidentiality and integrity.

```python
80:   def elections(request: LCHI):
81:       context = {'elections': Election.objects.all()}
82:       return render(request, 'elections.html', context)
83:
Listing 6.1: Function with common argument
```

43
Another generalization we are going to perform is concerning the objects stored in the server's database. These objects include the representation of the elections themselves, which include its id, the required public key, information regarding the administrator responsible for that election, among other fields. Maintaining the same coherence, we consider every field in a server-side object to be of HCLI. Listing 6.2 contains the formerly described scenario, in which the class `FenixUser` is a server-side one and therefore all of its fields will have the highest type annotation possible, HCLI.

```python
1: class FenixUser:
2:     name = "" # type: HCLI
3:     email = "" # type: HCLI
4:     status = "" # type: HCLI
Listing 6.2: Server's Object
```

In line 125 displayed in the listing 6.3 we can observe that the variable `data` will be populated through a JSON coming from a user. Since we already saw that the input coming from a request is always dangerous we will attribute every value of every key in the dictionary `data` the type LCHI. This case is also recurring throughout the code, the thought process being the same due to request always coming from a user source.

```python
125: data = json.loads(request.body.decode('utf-8')) # type: HCLI
Listing 6.3: JSON derived variable
```

In line 192 displayed in listing 6.4 we can observe that the variable `election` was obtained through one of the objects stored in the server, more specifically `Election`, which means the same logic applied to the server's object also applies to this variable.

```python
192: election = Election.objects.get(id = election_id) # type: HCLI
Listing 6.4: Server Object derived variable
```

In line 270 displayed in listing 6.5 we can observe that the variable `data` originates from filtering all of the objects of the class `Trustee` by its election `id`. Its value will be correspondent to all the `Trustee`'s for a certain election `id` in the format JSON. The `serialize` function that we witness is a Django function which has the purpose of converting Django objects into other formats, in this case JSON.
270: data = serializers.serialize('json', list(Trustee.objects.all()).
filter(election=election))) #type: LCHI

Listing 6.5: Serialized Server Object derived variable

In line 292 displayed in listing 6.6 we can observe that the variable reader comes from reading a file that was provided by the user. Since it was arrived via user input it will be regarded has LCHI.

292: reader = csv.reader(csvfile.read().decode('utf-8-sig'))
    .split('
'), delimiter=';') #type: LCHI

Listing 6.6: Variable derived from user input

In line 435 displayed in listing 6.7 we can observe that the variable questionData that has been derived from data which originated from a request provided by the user. However, since we are interested in comparing the various fields of the dictionary structure, we attribute its type as being a dictionary in which every value is LCHI. This will originate an illegal flow if any high value information is passed on to these values.

435: for answerData in questionData['answers']: #type: Dict[str,LCHI]

Listing 6.7: Dictionary type annotation

There were more type annotations made but they can explained following the same reasoning as the previously explained ones. Now that we have applied the correct type annotations to the code, we will dwell on the results from executing our tool in the next subsection.

Results

Reinforcing the fact that we were as conservative as possible when it came to annotating the variables, the tool managed to find 13 illegal information flows. Like the warning in listing 6.8, 12 more equivalent to it were generated.

There is an error in line: 131
The corresponding code is: election.name = data['name']

Listing 6.8: Warning example

All of the illegal flows discovered were regarding the security property integrity. As so that would mean that there was no secret information propagated into a public variable, which translates into no confidentiality breaches. This also translates into instances where unreliable information residing in
variables was considered reliable, which again resulted in the said leaks.

This would seem rather strange. Due to the high numbered nature of the code, one would expect to encounter illegal information flows regarding confidentiality. Specially taking into consideration that we have been as conservative as possible in our annotations to the code and have not used any type of downgrading, neither endorsement or declassification.

```python
131: election.name = data['name']
132: election.description = data['description']
133: election.hybrid = data['paperVotes']
134: election.startDate = data['startDate']
135: election.endDate = data['endDate']
```

Listing 6.9: Code containing information flow

In order for us to understand why this happened, let us closely analyze the code in listing 6.9. In a code with this size information will definitely be flowing between variables, like we can observe in lines 131 - 135, but every flow seems to have a common aspect. It always flows from the user input to the server’s variables. As we can observe in line 131 as a name of the election is taken from the input coming through the user and put in the field of an object in the server.

This phenomenon happens due to the nature of the program. Since the e-voting server is responsible for using unreliable and public information, which takes from the user, and create and store the result in a secret and reliable manner. This will result in no leaks when considering confidentiality but on the other hand will origin integrity related leaks since the unreliable information was passed on to a reliable variable.

However, this test case brought to light another limitation of this tool. It only deals with Python built-in functions, but in a world where frameworks proportionate more and more functionalities through their own built-in functions that represents a gap in our algorithm. Let us consider the following hypothetical scenario, a framework provides an alternative and exclusive method to print a variable. If the user tries to print a high level confidentiality variable by using Python’s function `print`, we will be in the presence of a leak and we will be able to detect it. But what if the user decides to use the alternative function? If it has the same functionality, or at least contains the original functionality from its Python counterpart, we will be in the presence of a flow that we cannot detect.
Chapter 7

Conclusions

This thesis presents a static analysis tool that is able to detect illegal information flows in Python. It allows for a high degree of flexibility since the programmer can define multiple security levels regarding confidentiality and integrity, while still remaining easy to use since it resorts to basic type annotations to the code. Even though in a conservative manner, it addresses third party libraries. If having access to the source code of such libraries it allows for the programmer to type annotate them as well which will reduce the possible false positives that we would normally have. This tool addressed both explicit and the more common implicit flows while providing the programmer useful feedback in order to keep his code secure. The flexibility that the programmer has in terms of attributing security annotations at his will comes with a price. It will be required of him to have security related knowledge specifically for his code in order to make the most out of this tool.

In the following sections we will display in which specific scenarios can our tool detect a leak as well as what could be improved in future work.

7.1 Achievements

This static information flow tool can address illegal flows in the following scenarios:

- Explicit flows
  - Assignment derived
- Built-in functions
- Implicit flows
  - Loop constructs
    - While constructs
    - For constructs
  - Conditional constructs
• **Third party libraries**

  – *With access to source code*: If we have access to the source code we can annotate it as well since thanks to MYPY it can detect leaks across files.
  
  – *Without access to source code*: If we don’t have access to the source code we can annotate what comes from the third party libraries. Even though it is less accurate than having analyzed the source code itself.

The current solution can now detect and warn the user of possible information flows in Python software, its performance depending on how accurate are the type annotations made to the target code.

### 7.2 Future Work

Even though the tool covers a number of possible information leaks it could still be improved in multiple aspects:

• **Flexibility regarding functions**: At the current state, the tool does not provide a way for the return value of a function to adapt to the computations performed in it, the programmer can only define a fixed value for the return of the function.

• **Lattice flexibility**: Implement the ability to generate a security lattice given any number of confidentiality and integrity levels, currently the tool requires confidentiality and integrity to have the same number of security levels.

• **Built-in behavior**: The behavior given built-in functions could be more specific given that currently it does not adapt to the behaviour of the function.

• **Increase code exposure**: Even though the tool has been exposed to a considerate volume of python code, it still wasn’t exposed to the gigantic dimension and near infinite scenarios that programs can have.

• **Implicit flow coverage**: Even though a decent number of implicit flows are covered, exceptions are not. The tool would highly benefit from being able to analyze try/catch constructs.

• **Framework limitation**: As it stands the tool can’t cope with functions provided by frameworks, being limited to analyzing the ones provided by Python.

In conclusion, this thesis provides not only an implementation of a tool that can address illegal information flows in Python but its algorithm, which may also be applicable to other programming languages.
Bibliography


50


